PROFESSOR: All right, I thought he would be. It's always a hazard putting up somebody who you know is going to lecture better than you do. But I hope that was a good change of pace. And, of course, he knows more about the subject of Reed-Solomon decoding than I or more than practically anyone else does, so -- did he have much time to talk about decoding or was the most -- yeah. Talked about the decoding algorithms, Sudan-type decoding algorithms. Yeah? OK. Good. Well, in fact, he covered so much that I don't have much left to do today.

I remind you, Ashish put out an email. There's a review session tomorrow for the midterm from 5 to 7 in a different room, 36-112. The midterm is Wednesday in this room. It starts at 9:05 to give you a little bit more time to work on it. It's closed book except that you're allowed to make up three normal size sheets of notes. You can write as small as you like. This experience has shown this is the very best way of getting you to consolidate what you've learned up to this point, is writing your own three-page precis of the course to date. So, yes?

AUDIENCE: [INAUDIBLE]?

PROFESSOR: What is the rule, both -- what?

AUDIENCE: Both sides.

PROFESSOR: Both sides. All right. Six sheets. If you can't write down the whole course in six sheets, then, well, you haven't been paying attention. OK. And you can bring calculators, but we don't expect they'll be useful. Please erase any erasable will memories if you do bring calculators. These are things we tell undergraduates. I guess they hardly seem necessary in a graduate course. Any other questions about the logistics of the midterm?

AUDIENCE: What is the material going to be for the midterm?

PROFESSOR: Everything that's been covered up to today. I won't hold you responsible for anything I've talked about today. So it's basically chapters one through eight. I really

will only hold you responsible for what we covered in class, which chapters one, two, and three were so brief, you can't say we really covered them. So really, chapters four through eight. Any other questions? All right. You'll have more chance as we go along.

So as I say, Ralf really covered the main things about Reed-Solomon codes, which are the crowning achievement of algebraic coding theory. They have a lot of favorable attributes. They are optimum, from an n, k, d point of view, in the sense that they're maximum distance separable. The parameters are as good as they possibly could be by this very basic Singleton bound. They have efficient algebraic decoding algorithms. And traditionally, this has meant hard decision bounded distance decoding algorithms like the Welch-Berlekamp algorithm or the Berlekamp-Massey match algorithm, which take hard decisions in, put hard decisions on symbols out.

But as Ralf suggested to you, now there are algorithms that can use soft decisions in, produce a list, at least, coming out, some indication of the reliability of each element of the list. So they've been softened up. Nonetheless, what are the negatives is, basically, that they work on bytes, or symbols, not bits. So if we have a basically binary channel, as we very frequently do, then they need something else to make them adapted to the binary channel, or even on the additive white Gaussian noise channel.

I don't think we've gone through the calculation. You could think of sending q or e symbols over the additive white Gaussian noise channel, but what would you need? You would need a q or e signal set. Now, if you pick that to be a q simplex signals set or a orthogonal or biorthogonal signal set, then that might work because the distance structure of such a signal set is like the Hamming distance structure in GF of q. In other words, it's approximately equally likely to make an error to any other symbol. The symbols are more or less equally spread apart in Euclidean space. Just as when we count distance in Hamming space, we give the same weight to each possible kind of error or each possible kind of symbol difference.

So if you did that, then Reed-Solomon codes might work well. That's one form of a concatenated scheme, which I will talk about in just a second. But if you tried to apply Reed-Solomon codes to a binary input additive white Gaussian noise channel, just translate the, say, 8-bit bytes into bits and send them one bit at a time, then the distance structures don't correspond at all well. It's much more likely to make a single bit error than an error that involves all eight bits or something. And for that fundamental reason, you'll find that performance is not so great.

And certainly, this is not the way to get to capacity all by itself unless it's married with something else. So that's something we always have to deal with in another applications, as I will talk about in a minute. We naturally do want to correct bytes, or symbols, or blocks. There are many bits. And then Reed-Solomon codes are just exactly what we want. They're optimal in any such situation.

And so what did I just say? They're not the way to get to capacity, although they are sometimes part of schemes for getting to capacity. And since getting to capacity in the additive white Gaussian noise channel is the theme of this course, then they play only a peripheral role in that theme. Nonetheless, they're great codes. They're in wide use. They're part of every semiconductor manufacturer's core library. You can buy (255,223) distance 33, 16 error correcting Reed-Solomon decoder in part of any chip package and it'll go at gigabits nowadays. So it's kind of the all-purpose error correcting code, with the caveat that you need to somehow get what you correct to be bytes rather than bits, OK?

So today I thought I'd cover the last two topics in chapter eight and maybe, to the extent that time permits, go back to chapter seven. But we'll see how we feel about that. Applications of Reed-Solomon codes. OK, basically, you can apply them wherever you want to correct blocks, rather than bits or packets or n-tuples or what have you. So one obvious application is, when you send packets through the internet, if you use Reed-Solomon codes across the packet, then you can correct errors that occur in individual packets.

This is not the way the internet usually works. The internet usually works by having

an error detection code within each packet, a cyclic redundancy check (CRC). You hash bits, whatever. You provide some redundant bits within each packet if -- when you receive the packet, you check to see whether the parity check checks. If there are parity check bits, then a very good rule of thumb is that the probability of not detecting an error for almost any type of error pattern is going to be of the order of 2 to the minus r. That's exactly what it would be if you had a completely random stream of bits. Random stream of bits with r redundant bits in a block, the probability that all the check bits are random, the probability that they're all equal to 0, is 2 to the minus r. All right?

But it turns out that's very good approximation for just about any error mechanism. So you pick r large enough. It's often been picked to be something like r equals 32. What kind of failure to detect does that give? 2 to the minus 32, which is what, about 2 the minus 9th, 2 to the minus 10th -- 10 to the minus 9th, 10 to the minus 10th. Is that low enough? It was certainly -- when I started in this business, that was considered just incredibly low, but now that we're sending mega packets per second, it doesn't seem quite so low anymore. And so 32 is a little bit on the sloppy side. There are going to be undetected errors that get through if you're spending 10 to the 9th, 10 to the 10th packets. But 32 or, nowadays, 64 would be a better number. Just like cryptography, the number of bits has to go up as technology goes up. But 32 is often used.

OK, but here's an alternative scheme. Suppose you coded across packets. So here is a, say, a 1,000-bit packet. And here's packet one, here's packet two, here's packet three, and so forth up to packet k, where we're going to use n, k, d equals n minus k plus 1, RS code over, it doesn't matter, GF of let's say 256, just for definite [INAUDIBLE]. Here I'm using older notation. This is the same as F-256. Or we sometimes write that as F 2 to the 8th.

OK, how are we going to do packet error correction? We're going to code crosswise in columns, all right? So these would be k. Let's make this eight-wide. Just pick off a bite at a time here. So each of these can be considered to be a symbol in GF of 256. And then we'll code n minus k parity check packets down here. So these are

redundant check packets. Encoded bytes at this point. They're not packets yet. But if we do that column-wise across here, we can fill in n minus k check packets and you can see the block length is quite huge here. So you might have a delay or latency problem in executing this scheme. Basically, you would have to send this whole thing before you could do error correction and then error correct the whole thing.

But how does error correction work? You send this. Perhaps some of the packets are erased or even received incorrectly. You would then use some error correction or erasure and error correction scheme, which Ralf talked about, for correcting the Reed-Solomon code. And you could correct up to d over 2 errors, or up to d minus 1 erasures, OK?

So this would be a very powerful packet error correction scheme. What are its costs? It's costs are quite a bit of redundancy. It doesn't fit the internet architecture very well. This is basically a point-to-point scheme. So when you're sending from a to b, you'd have to send b an additional n minus k streams. And if you're sending the same webpage, as you often do, to multiple people, you'd have to send this to everybody. And it also doesn't have any feedback in it.

The great thing about ARQ is that you only have to send the information that's necessary. You only have to restore the packets that are actually lost. You send them again. Whereas here, you're just kind of, a priori, deciding that you're going to send a great deal of redundant information down here to cover some kind of worst case, whatever you think the worst-case number of packets would be. If there are more that worst-case number or packets, then you could retransmit again, but you're retransmitting this huge block, all right?

So for that reason, this is not actually used, as far as I'm aware, in the internet. ARQ, wherever you have a feedback path, you should use it. That's a general moral. So ARQ is a wonderful scheme that is, on an erasure channel, that's a capacity-approaching scheme. Just on a bitwise basis, if you have a binary erasure channel with erasure probability p, and then the capacity of the channel is 1 minus

p, and you can reach that capacity if you simply retransmit each bit that was erased. As it's erased -- it's erased, you send it again.

And a quick calculation shows you that the average number of retransmissions is p. And so the average reduction from 1 bit per second is -- 1 bit per transmission is p bits per transmission, you can achieve 1 minus p with no coding at all, just by retransmission. So it's perfectly matched to the erasure channel. And the internet, especially with parity checks, CRC, within blocks, long enough CRC is basically a packet erasure channel. You really, in practice, hardly ever make a undetected -- let a packet with errors come through. OK? Any questions about this?

AUDIENCE: [INAUDIBLE].

PROFESSOR: The TCP/IP protocol just has a 32-bit CRC, I think, and if it fails to check, you ask for that packet again by packet number. OK? So Reed-Solomon codes, of course, are optimal if that's what you want to do. You couldn't possibly have a better code for coding across packets. But it's cumbersome in the packet environment. OK.

Let's talk about concatenated codes. I guess I'll do it at the same place. This is something -- I did in my thesis and last week I was at a tribute to Andy Viterbi, who invented the Viterbi algorithm, among other things. Of course, everybody knows that the Viterbi algorithm, when Andy invented it, he had no idea it was actually going to be a practical algorithm. It's become one of the super practical algorithms. He was just trying to prove a certain result, exponential error bounds for convolutional codes. And he thought the proof was easier to understand if you introduce this algorithm, which he describes perfectly. It is the Viterbi algorithm. But really, just a proof technique. Didn't even realize it was an optimum decoder.

And then somebody, not him, not me, but in his company, fortunately, realized that gee, this could be very practical. And that became a workhorse decoding algorithm. Very similarly, concatenated codes are something that I did in my thesis in 1965. And I was looking, basically, for a theoretical result. How could you approach capacity with an exponentially decreasing error probability near capacity but with only polynomial increasing complexity? And I looked into this very simple idea of

basically taking two codes and using one to correct the errors of the other. And I found that you could do that.

And I'd also never realized that this might actually be practical. I remember going around giving talks when I was interviewing afterwards that I was looking for an industrial position. I went down to places like Bell Labs, IBM labs, and did seminars talking about codes that were thousands of symbols long and everybody rolled their eyes and said, boy, this guy is not very practical. And to my surprise and everyone else's surprise, this became and still is a very practical way of getting high-performance codes. This, in conjunction -- well, I'll get into it, but this was, again, a workhorse coding technique and still is in wide use. In fact, as you will see, this contains some of the ideas that go into modern capacity-approaching codes.

But the idea is very simple. You can do it two ways, or probably many more than two. Suppose we have a basically binary channel. I'll just say a core physical channel there. The channel is what you can't affect as an engineer. So let's say we have a binary symmetric channel or a binary input additive white Gaussian noise channel or whatever you like. And then we could put on it an encoder and decoder for an n, k, d binary linear block code. OK?

Now, from the input and the output, what is this channel going to look like? At the input, we're putting -- basically, per block, we put k bits in. They're encoded into n bits. They're sent over the channel, received some n-tuple of noisy received symbols. The decoder tries to find the best code word. The code word is identified by k bits. Most of the time there'll be no error and these -- let's call this u in and u hat out. u hat is going to equal u.

But, of course, let's imagine this is a relatively short code. Let's imagine this is a maximum likelihood decoder or minimum distance decoder in the appropriate space. Then the complexity is going to be of the order of 2 to the k. And so you can't let k get very big. We'll find somewhat better ways to do maximum likelihood decoding. But if you're really trying to do optimum decoding, the complexity is going to go up exponentially at this stage with the number of information bits k. So this is

not the scheme that we're looking for that has only polynomial complexity. We know that we can choose codes like this that with optimum decoding get to capacity. That's Shannon's theorem. But we don't know how to decode them with reasonable complexity.

But, all right, here's a start. We could use a relatively modest length code that we can feasibly decode. And this is what we get. OK. So we have blocks going in. Block from an alphabet of size 2 to the k. Blocks coming out, alphabet size 2 to the k. Occasionally, we make errors. What's a good idea to then do? This is not very hard. This is why you should have been in this field in the '60s rather than the year 2005. There were lots of good ideas just laying around waiting to be picked up. Class? Anyone have a suggestion for how to make further improvement in this system?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Add another code. What should it be? Let's put an encoder here for what?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Everyone is extremely wide awake. An n, k, d Reed-Solomon code over what field? In principle, it should be over F2 to the k. Now, of course, if k is 64 or something here, you might not even go all the way to field of 2 to the 64 elements. You might divide it up again as we did in the packet error correction. But basically, the way this goes is you try to keep this as small as possible so that this can still work.

There's a tradeoff between the complexity of the two codes and therefore, the rates. The decoder -- here we have an algebraic Reed-Solomon decoder, which, in the simplest case, could just do error correction. Just take these k bits as a hard decision on u hat and try to correct any errors that occurred in this inner loop here, which we might think of this as being a 2 to the kre channel. So this whole thing together is a 2 to the kre super channel. Sorry if you can't read that, but you heard me say it.

So here's a channel that takes in 2 to the kre symbols, puts out 2 to the kre symbols and sometimes makes errors according to whatever the error probability you can

achieve here is. Or some of the discussion back when we were talking about hard decision with binary suggested that it would be better to pass along more reliability information here along with just the hard decision. And the very simplest kind of reliability information is if this guy really isn't sure, if either the noise is too large so he can't make any decision or if the noise leaves him exactly halfway between two code words so it's a completely ambiguous decision, or whatever criteria you have, this might make erasures.

And I'm sure Ralf told you how to do error and erasure correction with Reed-Solomon codes. Or this could be more likely that information just -- how likely was the code word you actually saw? And there are various ways of devising an approximate likelihood metric. You just do what makes sense that indicates that some words, you're absolutely sure of in this decoding. And other words, you don't know -- if you have one or more good candidates or you have no good candidates. So based on this reliability information, you can actually incorporate that. Did Ralf talk about reliability-based decoding? Soft decoding algorithms for Reed-Solomon codes? No? OK, well, that's a shame because he and Alex Vardy are the inventors of a very good class that's recently been improved.

So one of the knocks on Reed-Solomon codes is that, at one point, you might have said a minus can only work with hard decisions. And then, fairly early, it was figured out how to correct to erasures. But just in the last 10 years, we now have -- this is no longer true. We can have soft, meaning reliability-labeled inputs. And this is a plus. That means we can do soft -- and we have corresponding algorithms that can do soft Reed-Solomon decoding. They are, of course, more complicated than just straightforward error correction with hard decisions. But there's still polynomial complexity. They're maybe an order of slightly higher degree of n, but still polynomial in the block length or in k or d or whatever you're taking. And so we now have some such algorithms that do make substantial gains.

And, of course, if he didn't tell you about them, I'm not going to tell you about them. It's a rather specialized subject. But it's something that's really happened the last 10 years. And they've constantly been improved. The first ones, like one I came up with

called generalized minimum distance decoding, made a modest improvement. In dB terms, 1 dB, where there's 3 dB to be got. Now they're up to 2, 2 1/2 out of the 3 dB with these new soft Reed-Solomon decoding algorithms.

So anyway, if you pass some reliability information here, then you can improve the performance of this decoder. This decoder knows which symbols are reliable and it won't change in the decoding in which are unreliable or completely erased. Where there's no information about them, that actually helps you in the decoding. If you knew that you had s erasures rather than s errors, this helps you quite a lot. We can decode twice as many erasures as we can decode errors. And similarly, soft information in between can improve the capabilities of your decoder.

All right. So that's the set up. You can see why it makes a certain amount of sense. From a complexity point of view, what it does -- here, we're going to do maximum likelihood decoding, complexity order of 2 to the k. But we're not going to let k get very large, so this is feasible. Then we're going to let the code get very long, as we know we have to do from Shannon's Theorem, using this outer code. But now we know we have polynomial complexity Reed-Solomon decoding algorithms, even for the soft decision case.

So if we balance the complexity here, basically let this length be exponential and this length, as it is -- it's of the order of 2 to the k itself -- then we can get polynomial complexity overall, all right? The overall complexity is going to be of the order of a small power of 2 to the k. The overall block length is going to be of the order -- is going to be linear in 2 to the k. And therefore, you're going to get polynomial decoding complexity.

The other question is performance. And you can show that if you -- well, even if you don't use reliability information, you can get all the way to the capacity of this underlying channel here with exponentially-decreasing error rate for any rate below that capacity. Not as good an exponent as if you just used random codes per Shannon and coded for the channel, but nonetheless, you basically get what you want. So using two smaller, simpler codes to create one long code turned out to be

a good idea.

First of all, for proving this theoretical result, you could get exponentially-decreasing error rate for polynomial increasing decoding complexity. And within about 10 years, this is what people actually did. And, in fact, what people actually did -- I've got a block code here, but what people actually did was they used a short constraint length convolutional code here, which we'll be talking about after the break.

Convolutional codes tend to have a better performance complexity tradeoff, particularly in soft decision situations than block codes do. So in a practical system, it's almost always better to use a convolutional code than a block code. We'll discuss the reasons why.

So this convolutional code here, there is a maximum likelihood sequence detector for the convolutional code down here, which is the Viterbi algorithm, which you've probably all heard about. And this is very practical as long as the constraint length, k, now becomes the constraint length, nu, of the code doesn't get too large. So for about 20 years, the standard for space communication was this was a (255, 223, 33) Reed-Solomon code over F-256. This is a constraint length 6, rate 1/2 convolutional code. This is a 64-state Viterbi algorithm.

So what's the -- again, if we have a look at the error channel, here is a stream of bits coming along. After encoding and decoding, what we typically see is bursts of errors, all right? So you get a burst that starts -- in convolutional codes, it can start at a random time, end at a random time. You might get another discrete error event over here. In other words, most of the time, the decoder is decoding properly. What you're receiving is what you transmitted in a streamwise fashion. The bits just are transmitted forever. But every so often, the decoder gets off on the wrong track, puts out errors for a while, then gets resynchronized, and decodes correctly again. Is my picture clear? I'm using words, mainly, to tell you what it represents. OK.

So now, basically, what you want to do is, just as we did in the packet communication case, you want to send multiple parallel streams, up to 256 of them, such that in this stream, you're going to get independent error blocks. And this one,

you're going to get errors occurring somewhere else. They're going to have a typical length. You can very well estimate what the typical length of error events is, but they occur randomly. And now you use the Reed-Solomon code, coding across this, and that will basically give you this picture over here.

How would I get this? One way is by using a block interleaver, which is described in the notes. A block interleaver, you basically take a long block from the convolutional code and you just encode and transmit along here and ultimately decode. And after a very long time, you come back and start transmitting the second row just as part of the same stream. And then the third row and so forth. And if the block length n, here, is long enough, the errors in different rows will be effectively independent, all right?

So again, you wind up with very long blocks in this scheme, but analytically, it works. And in practice, this is roughly what's done, except there is such a thing as a convolutional interleaver, which again, operates streamwise and actually performs better in practice. But I won't take the time to go into this because we're not going to really be getting into things like interleavers. Do you see how this works? OK. So it works well theoretically. It gives us the theoretical result we want. Works well in practice.

As I say, this was the standard for space communications during the '70s and the '80s. Around the end of the '80s, they upgraded this. First of all, they replaced this by a nu equals 14, rate 1/6, lower rate convolutional code, so this now becomes a 2 to the 1fourth state Viterbi decoder. They built this huge rack at JPL that had 8,192 parallel decoding units, add-compare-select units in the Viterbi algorithm, to decode this thing. But, of course, they only need to do that in one place, in the space program. Everything comes back to JPL. So they built this BVD, Big Viterbi decoder. And first, they used it with this code and then they souped up the Reed-Solomon code. And by about 1990, 1991, they got this thing operating up within about 2 dB of the Shannon limit, OK? Which was considered a heroic feat at the time, or even now.

So that kind of tells you, in the terms we've been using in this course, just what you can get out of this kind of approach. At that point, that was considered coding is dead. There's really nothing more to do. We're within about 2 dB of the Shannon limit and it's inconceivable that we could actually get any closer than that. Second half of the course will, of course, tell us how we can get within 0.0045 dB of the Shannon limit. But that was the state of the art as of about 1990, 15 years ago. Questions? Comments? OK.

All right. I'll just mention a third application is burst error correction, which really, we've already been talking about. Suppose -- let's let this be a naked channel now. Suppose you're transmitting with an interleave scheme like this. And now, in the same way -- it's a radio channel or something and every so often, you have an outage just due to fading, or a thunderstorm, or somebody turns on his vacuum cleaner, or whatever. Every so often, as you transmit, you're going to get bursts. You don't have independent, identically-distributed errors. The character of the channel is such that you see bursts of errors, which is precisely what you do in this super channel here. Every so often, you see a bunch of errors if you have the original transmitted scheme to compare it to.

All right. In exactly the same way, you can code -- with something like this block interleaver, you can code across channels and correct the bursts with Reed-Solomon codes. And here, it's quite easy to show that, because Reed-Solomon codes have optimal parameters n, k, d, that Reed-Solomon codes do the best possible job of correcting burst errors that you possibly could do, given some specification for how long the bursts could possibly be and the minimum guard space between bursts. This is classical stuff. Reed-Solomon codes are optimum burst correctors if you describe the burst channel by max burst length in a minimum guard space. Is that too brief? Probably is. Give you some impression of what we're talking about.

So Reed-Solomon codes are optimum for burst error correction, which, most real channels are bursty. The only non-bursty channel that we really have is the additive white Gaussian noise channel, ultimately. That is the channel in space

communications, which is why you can apply a nice, textbook-type code like this to the space channel. For any other channel, typically, you have to use some kind of interleaving to break up the bursts so that you can -- this simply is no information over a long time. If you tried to encode crossways on this, you'd have to have a code that's long enough to see the average burst's behavior. We'd make the burst have an ergodic model. The way to do that in practice is to code across it so you get independent snippets of widely separated bursts which can be assumed to be independent. Yeah.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: No different, really.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah, in space channel -- all right. Here, I'm talking about power limit. That's a good question. But if you think about the band-limited regime, let's suppose -- so this would be some [? MRE ?] channel. You would use some multilevel signaling scheme like QAM with a large signal set or QPSK or something like that to get a basic transmission with a discrete signal set that doesn't lose too much over sending Gaussian over the channel. Again, this will be later in the course. We'll talk about this in detail.

Other than that -- well, all right, so now this is an MRE channel. If M is large enough, then you could apply Reed-Solomon codes right here. But typically, it's not. M is like 16 or 64. And so Reed-Solomon codes would still be too short. You would not be able to get long enough codes over such a channel. So a similar kind of thing can be done. And the history of bandwidth-limited coding pretty much parallels that of power-limited coding.

What corresponds to block codes here is lattice codes for this channel. What corresponds to convolutional codes is trellis-coded modulation. And again, once you handle the basic physical channel with these codes that are well matched to that channel but whose complexity increases too rapidly if you make them long because

of exponential decoding, then you can always clean up whatever errors you make.

You can think of this as a cleanup function out here. Typically, the rates out here are very high. You don't need much redundancy. It doesn't cost too much in overall rate. But even with very high rates, like 223 over 255, you can do a rather large amount of error correction. You can correct up to 16 errors or 32 erasures with us channel. So this is a cleanup function.

You basically work very hard in here, get the best error probability you can, which -- it doesn't have to be too low. 10 to the minus 2, 10 to the minus 3. And then you use this to clean it up and that drives the error probability down to 10 to the minus 12 or 10 to the minus 15 or whatever you really want. But the principles are the same in power limit and band limit. But nice question. Any other questions? OK.

So as a result, Reed-Solomon codes are the general purpose code that has emerged from algebraic coding theory. A good example of burst error correction is, you know on your CDs, there is an error correction scheme incorporated such that if you scratch the CD, it still works. How does that work? It could only be because there's some error correction in there.

What, actually, is the error correction scheme? I'm not sure I know what the latest and greatest is, but generally, the error correction schemes for magnetic memory recording like that have used Reed-Solomon codes -- have used two Reed-Solomon codes in interleave fashion like this. So the across code is Reed-Solomon. The down code is Reed-Solomon.

As a result, if you make a scratch through the disc, you will cut all of these codes, but you will cut each one in a small enough place so that you can correct it. So you can think of this as a actually a physical laying down of Reed-Solomon codes on the two-dimensional surface of the disc in such a way that -- well, even if your scratch came right across one whole channel here, then, of course, this decoder would be dead. But you still have somebody decoding in this direction to pick up the errors. And so that's why you can completely abuse these discs and they still work.

And when you think about what's actually going on, think of the transfer rates from a CD to your computer. It's up in the hundreds of millions of bits per second. Maybe its gigabits by now. Perhaps it is. And you've got this Reed-Solomon -- these are non-trivial Reed-Solomon decoders. These are like this 16 error correcting decoder out here. And so these are working away at that data rate, capable of doing decoding. And they're cheap enough so that they're just buried in part of one of the integrated circuits in your disc player circuitry.

So that'd give you a physical sense for just how much error correction power and speed you can get out of this class of code decoder, OK? So wherever you want an extremely powerful code and you're a little bit less concerned with getting to capacity, Reed-Solomon code is the way to go. It's absolutely part of your toolbox, OK?

The last topic in chapter eight is BCH codes, which are binary codes. So they're binary linear block codes of the class we've been discussing, comparable with Reed-Muller codes. Reed-Muller codes were discovered in 1954 in two separate papers by Reed and by Muller in two separate points of view. And there are a million points of view you can have for Reed-Muller codes. We've used yet a third one.

BCH stands for Bose and Chaudhury and Hocquenghem. Bose and Chaudhury were both professors in the US. They wrote a paper about 1960 introducing Reed-Solomon codes, which, by the way, and introducing BC codes -- Bose-Chaudhury codes -- Reed-Solomon's paper was in 1960 also. These were completely independent of each other. And then this fellow named Hocquenghem in France, whose name I can't and few people can pronounce, turned out he had actually invented these codes in 1959 in a paper that unfortunately was in French, so few people read it.

But anyway, in honor of all three of these people, these are called BCH codes. These fit right into the main theme of algebraic coding theory at that time. Maybe stimulated it because these turned out to be cyclic codes, or at least as they were

originally introduced. And so when I was in school, which was in the early '60s, these were considered the greatest thing going. People were interested in binary codes, algebraic codes because they are slightly better than Reed-Muller codes.

Now, I would say, there's sort of been a swing back. Shu Lin gave a paper about 10 years ago that said Reed-Muller codes are not so bad. In fact, in terms of performance versus complexity for certain kinds of decoding schemes, trellis-based decoding, Reed-Muller codes are better than BCH codes. So from a more modern prospective, we're more interested in Reed-Muller than we are in BCH. Nonetheless, you'll hear about these a lot if you read the literature and so you ought to know what they are.

OK. Conceptually, what are they? Since we already know about Reed-Solomon codes, the way that I'm going to characterize BCH codes is as subfield subcodes of the Reed-Solomon codes. Now, if we have a Reed-Solomon code over F2 to the 8 or something, the field with 256 elements, or any power of 2, contains a subfield which is just 0 and 1, all right? All fields contains 0 and 1. If the field has characteristic 2, then 0 and 1 all by themselves are the prime subfield. So it's possible that there are certain words in the Reed-Muller code -- sorry, in the Reed-Solomon code that are completely made up of 0's and 1's.

For instance, the all-0 word is certainly in the Reed-Solomon code, so we have, at least, that one that's made up of all zeros. Almost always, we find that the all-1 word is a code word. So we have a code with all 0's and all 1's. And there might be more of them. And that's actually very straightforward, to find out how many completely binary words there are in a Reed-Solomon code.

So it's the binary subfield. We have F2 is a subfield of F2 to the M for all M. OK, so a subfield subcode, if we have an n, k, d equals n minus k plus 1 Reed-Solomon code, then the set of all BCH code is the set of all binary code words, code words that just use the 0 and 1 of the code alphabet. Call this c in C. So we call this C prime.

OK, so what are its parameters going to be? Its length is still going to be n, right? All

the code words here are n-tuples, so these have got to be n-tuples. We don't know its dimension. Let's just call it k prime. That's how many such code words are there? We're going to have to do some combinatorics to find that out. And what's the minimum distance going to be? Anybody?

AUDIENCE: [INAUDIBLE].

PROFESSOR: In general, well, it's certainly not going to be less than because, by definition of the minimum distance, the minimum Hamming distance between any two code words in this code is d, so we're just looking at a subset of the code words in this code. And the minimum squared distance has to be at least d. There's an example given in the notes. If we let c be the 4, 3 -- I think it's the 4, 2, 3, over F4, then C prime turns out to be the 4, 1, 4 code over F2.

In other words, the only two code words in this code that are all binary are the all-0 and all-1 sequences. All right? So we actually get a minimum distance of 4. It has to be at least 3 in this particular case, so here's an example where it could be greater than 3. Or if there are cases where the subcode just consists of the all-0 word, then that, by convention, has an infinite minimum distance. So distance could be greater, but it never can be less than what you started with.

On the other hand, the dimension is clearly going to go down. If it were still equal to what it was, then we'd have a binary MDS code and we know we don't get binary MDS codes. So in general, k prime decreases quite a bit. In this case, it just decreases by 1. And we have to find out how it decreases.

OK. I'm a little uncertain as to how much of the derivation -- so we need to find --

AUDIENCE: Is that [INAUDIBLE] at all points of the field, is it?

PROFESSOR: No. One of the characterizations you have of Reed-Solomon codes, I assume, was that they're the set of all polynomials of length n that evaluate to 0 at some consecutive set of n minus k powers of alpha. Did you see anything like that?

AUDIENCE: Yes.

PROFESSOR: All right. In other words, they're -- let me write that down because RS code is the set of all multiples f of x, g of x, of degree less than n where of g of x equals the product of x minus alpha to the i for i equals 0 to n minus k minus one or something like that. This is called a generator polynomial and -- so what does this mean? In other language, all polynomials of degree less than n that have roots at 1 alpha, alpha squared, or so forth. It's better to make this i equals 1 to n minus k. You can also make it minus i. There are lots of details in here. But I'll just ask you broadly to recall that characterization. This is a --

AUDIENCE: I have another question. [INAUDIBLE] are not linear?

PROFESSOR: OK, let's ask that. So we've got a set of binary n-tuples, OK, defined as the subset. So what do we need to do to see whether it's linear or not?

AUDIENCE: [INAUDIBLE].

PROFESSOR: We just check the group property. Is it closed under addition? Suppose we take two of these code words we add them together, component-wise, vector form. Seems to me we get A, that gives us another code word in the Reed-Solomon code because it's closed under addition, and B, by the addition rules, it's going to be another binary code word, all right? So it's another binary code word in the RS code, therefore, it's in C prime. So that's proof that, in fact, this code is linear.

AUDIENCE: What about [INAUDIBLE]?

PROFESSOR: That's trivial. For a binary code, we only have to consider two scalars, 0 and 1.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: But now I'm asking whether this binary code is linear. There's the code over F2. Yes? You had a --

AUDIENCE: Same question. The base field is F2 to the x, so if you add 1 and 1 there, it wouldn't be just 0. But I guess if you restrict the field to F2, you have to check everything again, don't you?

PROFESSOR: No, because we said F2 is the subfield of F2 to the M. And what we actually showed was that if you just stay in the prime field, the integers of the field, then you get exactly the same addition and multiplication rules as you do in the prime field. So in fact, in a field of characteristic 2, we always have 0 plus 1 is 1, 0 plus 1 plus 0 is 1, 1 plus 1 is 0. That last one is the only one you really have to check. So that accounts for the definition of characteristic, and that's the reason we can always use just plus signs, because subtraction is the same as addition.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Yeah. We showed -- well, very simple proof. The order of a subgroup has to define the order of the group it's in. The additive group of the prime field has order 2 to the M, so it has to be some divisor of 2 to the M, which is a prime. I think we proved that another way, but that's a very simple proof. OK, great questions, because it shows you're thinking about how this relates to what you learned back in chapter seven. But at this point you've kind of got a little vague about chapter seven, what's really in there. By Wednesday, you're really going to know chapter seven well. Let me tell you. OK. Good. Thank you for the help. Because these are all things that I should have just said up here, but they come out much better, actually, if they come out in response to questions. OK.

So here's one characterization of a Reed-Solomon code which apparently you did see. It's the set of all multiples of some generator polynomial. The generator polynomial is described as this polynomial of degree n minus k whose roots are precisely alpha up through alpha to the n minus k. You did say that? OK. So similarly, the BCH code -- I'm just going to assert this just to show you how it goes. It's the set of all multiples of g prime of x, which is the minimal degree binary polynomial with the same roots. Now, how can a binary polynomial -- alpha to the n minus k.

How can a binary polynomial have non-binary roots? Well, it clearly can. Let's consider g of prime of x. It's an element of F2 to the x, the set of all polynomials over F2, which is a subset of the set of all polynomials over 2 to the M for the same

reason. F2 is the subfield of F2 to the M, so some of the polynomials over F2 to the M have purely binary coefficients, all right? So that's the sense -- that's how we can get between these two things.

So some binary polynomials have roots in GF of 2 the M let me give you an example. Let's consider x squared plus x plus 1 as a polynomial in F4 of x, which is certainly is. It's a polynomial of degree 2 and F4 of x. What are its roots in F4 of x? Does it have any roots in F4 of x? Does it have any roots in F2 -- sorry, I'm asking whether it has any roots in F4. Now I step back and ask if it has any roots in F2. So how do we do that? We just ask if F of x is 0 for x equals 0. We get 1. So it's not a root of 0. Substitute 1, we get 1 plus 1 plus 1. 1 is not a root F of x.

All right. Well, we have two more candidates. How about alpha? For alpha, we get F of alpha equals alpha squared plus alpha plus 1. And if you remember the addition table of F4, however it's constructed, this equals 0, OK? 0, 1, alpha, alpha, squared. 0, 1 alpha, alpha squared. 0, 1, alpha, alpha, squared. This is plus. 1, alpha, alpha, squared. 0, 1 plus alpha is alpha squared. 1 plus alpha squared is alpha. Any of these is telling you the same thing. 0, 1, 0, 1 alpha, squared alpha.

So we conclude from that that 1 plus 1 plus alpha, which is alpha squared, plus alpha squared is 0. So alpha is, in fact, a root of this. And what about F of alpha squared? Alpha squared squared is alpha fourth, but we can reduce that mod 3 -- the exponent mod 3. So this is alpha plus alpha squared plus 1 and that is equal to 0. So alpha and alpha squared are roots of this polynomial.

There's a lovely fact proved in the notes, which is that every element of -- facts. Every element of F2 to the M is a root of x 2 to the M plus x.

OK, did you see something like this last time? I see some shaking of heads. OK. And how many roots could there possibly be of x 2 to the M plus x by the fundamental theorem of algebra?

AUDIENCE: 2 to the M.

PROFESSOR: 2 to the M. So that means that x 2 to the M plus x -- and for every root, it's got to

have a factor of the form x minus, or in this case, plus beta. So this must have a complete factorization of the following form. x 2 to the M plus x is simply the product of x minus beta. All right. In fact, you can see that 0 a root, obviously. 1 is a root, obviously. And we've checked up here. Let's see, what are we interested in? Here's another example. x to the fourth plus 1, f of -- if we have g of x equals x to the four -- sorry -- plus x, then g of alpha is alpha to the fourth plus alpha. But alpha to the fourth is equal to alpha, so that's 0. g to the alpha squared equals alpha to the 8th plus alpha squared. Alpha to the 8th is alpha to the fifth is alpha squared, so that equals 0. So, in fact, that's true for that case.

Every irreducible polynomial g of x and F2 of x whose degree g of x divides 2 to the M -- is that right? This may be 2 to the M minus -- I think it's whose degree divides M -- is a factor of x to the 2M plus x in F2 to the x. OK. So let's give you an example again. Example M equals 2. This means, what are the irreducible polynomials whose degrees divide 2? The prime polynomials of degree 1 certainly divides to our x and x plus 1. Do they divide x to the fourth plus 1? Yeah, they do.

How many prime polynomials are there of degree 2? There's only one, x squared plus x plus 1. And if you divide it out into x to the fourth plus 1, it goes evenly. It turns out that what this implies is that this is actually an if or only if. A polynomial g of x divides x to 2 to the M plus x if and only if g of x has a degree which divides M and g of x is irreducible. So this implies that x to the M plus x factors in F2 of x into the product of all irreducible polynomials whose degrees divide M. And I think you did a homework problem that used this, even though it was never proved in class. So you at least read about it.

Again, for this example, that should mean that x to the fourth plus 1 equals x times x plus 1 times x squared plus x plus 1. Is that true? Pretty easy to see that it is. x plus 1 times x squared plus x plus 1 is just x cubed plus 1. Multiply by x -- I'm sorry, this x fourth plus x. How many times have I done this? Should be x to the fourth plus x. Yes. OK, it's kind of miracle when you first see it. This works over any field for any PNM again, lest you get this kind of factorization.

OK, but now, comparing this to this, what does that mean? That means this is one factorization. This is in -- we can do a complete factorization in F2 to the M of x. We can only do complete in the sense that we reduce it to a product of degree 1 polynomials. In F2 of x, we can't do a complete factorization. This is similar to, over the complex field you can always completely factor a polynomial into one-dimensional polynomials of this form, where beta is the set of roots of the polynomial. Whereas over the real field, you might have to have some degree 2 polynomials in there.

This is exactly analogous. You can't get complete factorization over this subfield. But what does it mean? Here we have a polynomial of degree 4, has 4 roots. We factor like this. So each of these factors must contain a number of roots equal to its degree. So what are the roots? This is the polynomial that has root 0. This is the polynomial that has root 1. As we just showed, this is the polynomial that has the roots alpha and alpha squared, OK? So it's always going to turn out that way.

So you can group the field elements into subsets where each subset is the set of roots of some irreducible binary polynomial of some degree that divides M, OK? These are called cyclotomic subsets. Whence the name of Elwyn Berlekamp's company, Cyclotomics. All right. So this is where, I think, the theory gets just absolutely lovely and unexpected, at least when you start in. And people who have a taste for these things, I encourage you to read a real book on the subject. Berlekamp's is one of the most elegant. But there are lots -- because it's an elegant subject, there are lots of elegant treatments of it. All right.

So remember what we were trying to do in the first place. We were trying to find the dimension of BCH codes. So where are we now? We've defined a BCH code as the set of all binary polynomials with roots alpha up to alpha to the n minus k where n and k are the parameters of the parent Reed-Solomon code. And alpha is in the parent field. So alpha in F2 to the M.

OK. One final fact, I guess, we need to know here. All right, if alpha is a root -- now I'm talking about any field -- then alpha squared is a root of a binary polynomial.

And again, this is due to a very lovely fact, which is that squaring is linear in fields of characteristic 2. Why is squaring linear. A little side calculation. a plus b squared equals a squared plus 2ab plus b squared but characteristic 2. We can wipe that out. So in general, f of x quantity squared is f of x squared. And that's why, if f of x evaluates to 0, then f of x squared must evaluate to 0 since 0 squared is always equal to 0.

All right. So alpha is a root if and only if alpha squared is a root. Is that true here? Yes. Alpha alpha squared. Well, is it also true if we squared alpha squared? What is the square of alpha squared? It's back to alpha, all right? So these sets now turn out to be cyclotomic cosets, they're called, which is the set of all elements and their squares. We get alpha, alpha squared, alpha fourth, alpha eighth. And then, at some point, it cycles. Why? Because what does alpha to the 2M equal?

AUDIENCE: [INAUDIBLE].

PROFESSOR: No. Alpha. Right. Alpha to the 2M minus 1 is 1, so alpha to the 2M is alpha. So it comes back to alpha again because of that. So we get a finite number of roots which, of course, are equal to the degree here. In some cases, it's fewer. It's whatever the degree is. It's d, which divides M.

So from this, we simply start -- suppose we want alpha to be a root. Let's start with high rate codes. I think here in the example, in the text, I used F16. F16 is going to have -- the elements of F16 are going to be the roots of all the degree 1, 2, and 4 polynomials because M is 4 and these are the degrees that divide 4. So again, we get x and x plus 1, which have roots 0 and 1. We get x squared plus x plus 1.

What are the roots of x squared plus x plus 1 and F16? There are only going to be two of them and they've got to have the property that if we start with the first one, let's call it beta and beta squared. Beta fourth has got to equal beta because we only have two of them. That more or less forces beta to be alpha to the fifth or alpha to the tenth. Let's see if that works. We get alpha to the fifth, alpha to the tenth. Square that, we get alpha to the 20th. But we now take the exponents mod 15 and we get 5 again, all right? So those are the two roots of x squared plus x plus 1 in

F16. Obviously, I'm not -- I guess that was a legitimate proof, but it was certainly quick.

OK, there are three polynomials of degree 4. By the way, I think you can imagine how, given this result, we can now enumerate the polynomials of each degree. We now have an expression. There are two irreducible polynomials of degree 1. So how many more of degree 2 is it going to take to make the overall degree 4? One of them. So we're looking for a single -- there must be a single irreducible polynomial degree 4. Similarly over here, all right, we've accounted for four of the elements of F16. The only other possible degree we could have is 4. So how many irreducible polynomials are there of degree 4 over in F 2 of x? Must be three of them. And it always works out.

So, from this you can get a closed-form combinatorial formula again in terms of the Mobius inversion formula. By doing a sieve or by some other method, you can find out that the three irreducible polynomials of degree 4 are those three and the corresponding roots for this one -- if this is the irreducible polynomial you use to define the field, then alpha is a root of it and you're going to get alpha, alpha squared, alpha fourth, alpha to the eightth. Alpha to the eightth squared is alpha to the 16th, but that equals alpha, OK?

So these are the roots of that equation. Here you get -- sorry, x fourth plus x third plus 1, since that's basically the reverse of this, turns out you get alpha to the minus 1, which is alpha to the 1fourth. Square that and you get alpha to the minus 2, which is alpha to the 13th, or equivalently, alpha to the 28 -- reduces to thirteen. This becomes alpha the 11th, alpha to the seventh. and then alpha to the seventh gives you alpha to the 1fourth again.

AUDIENCE: [INAUDIBLE] same thing about taking the square [INAUDIBLE]?

PROFESSOR: Correct.

AUDIENCE: --from the [INAUDIBLE]?

PROFESSOR: Because you go around in a circle. So it's a finite set. So you can obviously divide by

2. So there's one cyclotomic coset. Here's the second one. And then the third one must be whatever's left over. What's left over? We don't have alpha to the third yet. Alpha to the third, alpha to the sixth, alpha to the 12th, and then alpha to the 2fourth, which is alpha to the ninth. And alpha to the 18th is equal alpha to the third again.

OK, what's the moral of this? We want to start off with -- let's just take a look at the Reed-Solomon, the very first one. We just want n minus k to be 1 and we just want to have root alpha in here. Well, if we get root alpha, we're going to get root alpha squared, alpha fourth and alpha eightth along with it. If we want a binary polynomial with root alpha, it has to this one. And it's going to have four roots. All right.

So for root alpha, we're going to get x fourth plus x plus 1 as our generator polynomial. And it's actually going to have roots alpha and alpha squared. So it's going to be the same one as you would get if you wanted alpha and alpha squared. That means n minus k is already up to 2, which means your distance is already up to 3. Distance is n minus k plus 1. So this leads to a code with n equals 16 -- is that right? Yes. BCH codes are always defined to be one shorter. We're going to have four redundant bits, so k equals 12 and distance equals 3.

I think that's not quite right. When we're doing BCH codes, they're always defined just on the non-zero elements of the field. So we start with the shorter Read-Solomon code of length q minus 1. 15 in this case rather than 16. And 0 never appears here as one of the possible roots. So let me just do that. I apologize I haven't carried through all the possible steps. So we're getting all of the polynomial multiples of x fourth plus x plus 1 of degree less or equal to 14 rather than 15. So it's n equals 15. n minus k is going to be 4. So this is 11. And the distance is going to be greater than equal to 3. In fact, it is 3. So it's a 15, 11, 3 code, which is, in fact, the Hamming code. Yes?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh, I'm sorry. Wasn't this fun? You're supposed to do that, Ashish.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK, well, so you can see how it goes. So this gets us both roots alpha and alpha squared. If we want alpha third and alpha fourth, we already have alpha fourth from this polynomial. So all we need is this one, which gets us another. And that gives us an n equals 15, k equals 7, d greater than or equal to 5, which, in fact, is a 15, 7, 5 code. So we get additional codes. There's a table in there that shows you that most the time for short block lengths for distances which are equal to a prime power, we get the same parameters as Reed-Muller codes. For lengths 64 and higher, sometimes we do a little bit better.

In addition, we get a lot more codes for every, basically, odd distance when you take the length odd or even distance if you take the length even. And that's definitely an asset, that you have a larger number of codes to choose from. So there's no doubt that in terms n, k, d, BCH is the larger and, for longer block length, slightly better class than Reed-Muller codes. But as I said before, in terms of performance versus complexity, Reed-Muller might still be the ones you want to use. OK. Good luck in the review and good luck in the exam. I will see you Wednesday.